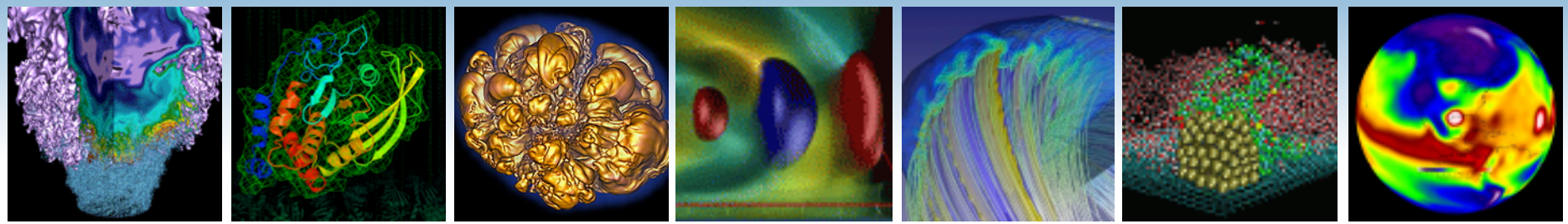
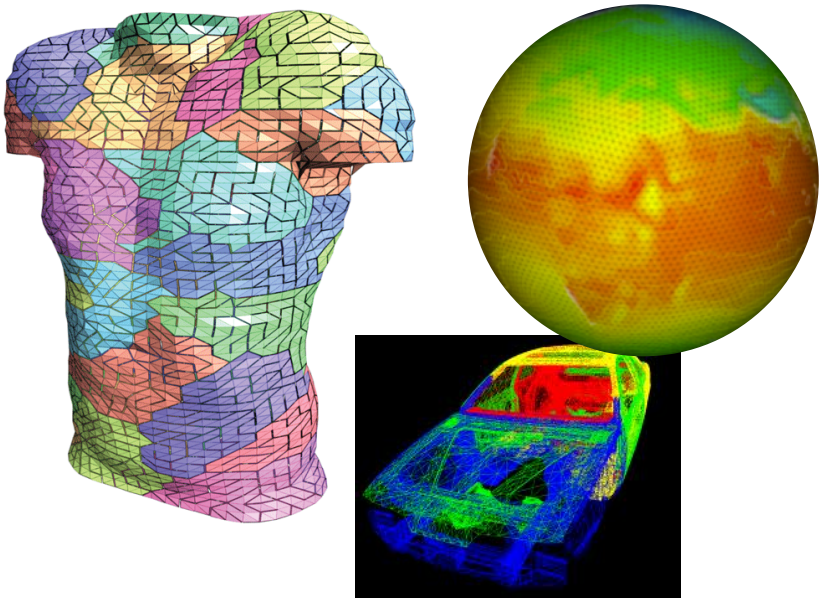


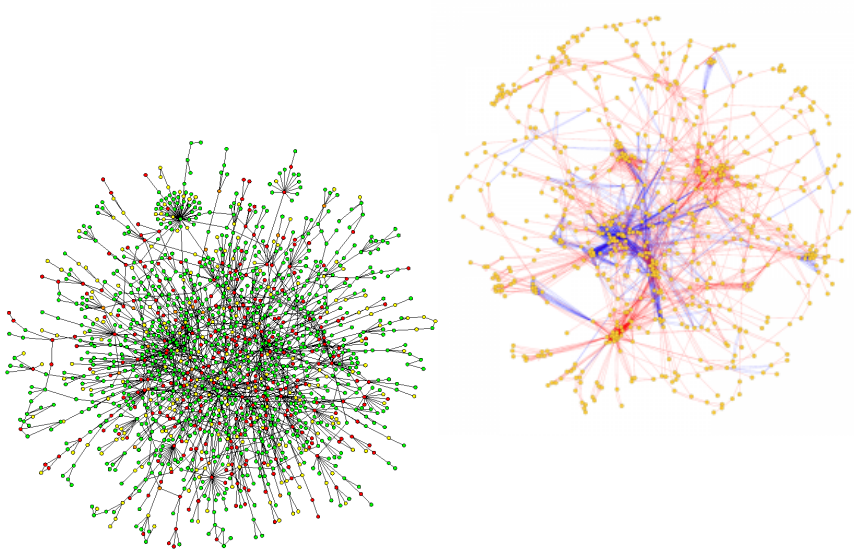
Supporting Irregular Applications with Partitioned Global Address Space Languages: UPC and UPC++

Kathy Yelick
Lawrence Berkeley National Laboratory

With results from the DEGAS and UPC groups

Programming Challenges and Solutions





Message Passing Programming

Divide up domain in pieces

Each compute one piece

Exchange (send/receive) data

PVM, MPI, and many libraries

Global Address Space Programming

Each start computing

Grab whatever you need whenever

Global Address Space Languages and Libraries

5-10% of NERSC apps use some kind of PGAS-like model

UPC

BERKELEY LAB
Lawrence Berkeley National Laboratory

Shared Memory vs. Message Passing

Shared Memory

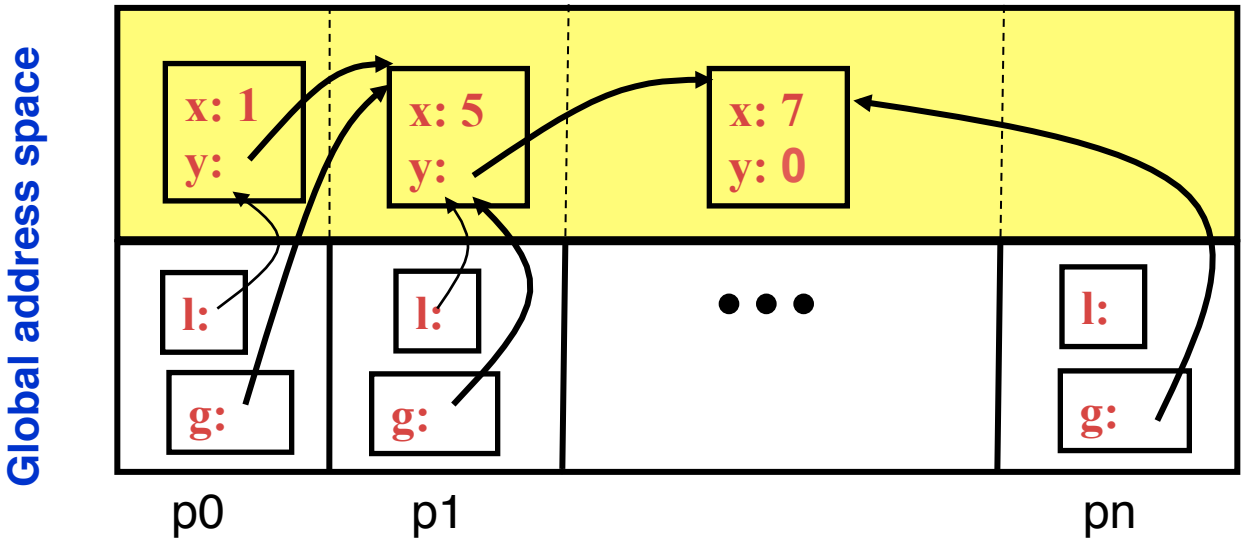
- Advantage: Convenience
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- Disadvantages
  - No locality control
  - Does not scale
  - Race conditions

Message Passing

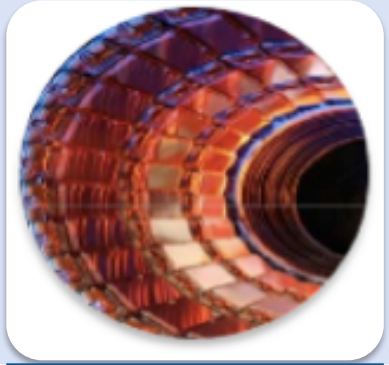
- Advantage: Scalability
  - Locality control
  - Communication is all explicit in code (cost transparency)
- Disadvantage
  - Need to rethink data structures
  - Tedious pack/unpack code
  - When to say “receive”

PGAS: Partitioned Global Address Space

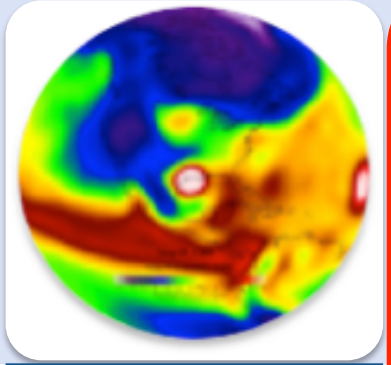
- ***Global address space:*** thread may directly read/write remote data
  - Hides the distinction between shared/distributed memory
- ***Partitioned:*** data is designated as local or global
  - Does not hide this: critical for locality and scaling



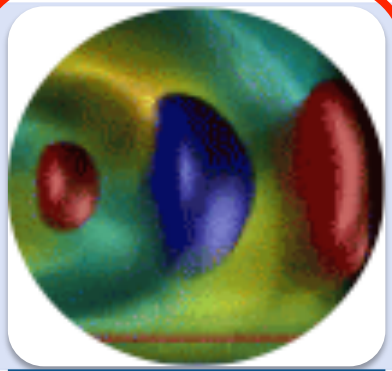
Science Across the "Irregularity" Spectrum



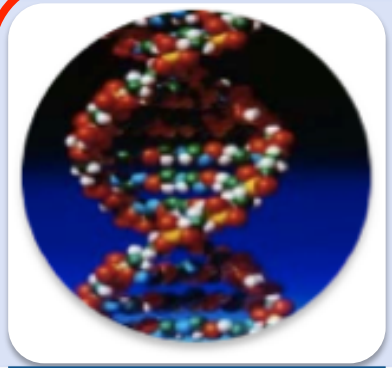
Massive Independent Jobs for Analysis and Simulations



Nearest Neighbor Simulations



All-to-All Simulations



Random access, large data Analysis

Data analysis and simulation

Hello World in UPC

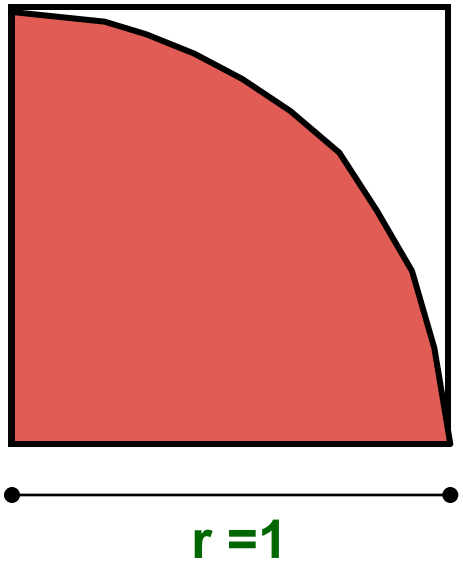
- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the a few UPC keywords:

```
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - π = 4*ratio



Pi in UPC

- Independent estimates of pi:

```
main(int argc, char **argv) {
  int i, hits, trials = 0;
  double pi;

  if (argc != 2)trials = 1000000;
  else trials = atoi(argv[1]);

  srand(MYTHREAD*17);

  for (i=0; i < trials; i++) hits += hit();
  pi = 4.0*hits/trials;
  printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” separately

Helper Code for Pi in UPC

- Required includes:

```
#include <stdio.h>
#include <math.h>
#include <upc.h>
```

- Function to throw dart and calculate where it hits:

```
int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
       return(1);
  } else {
       return(0);
  }
}
```

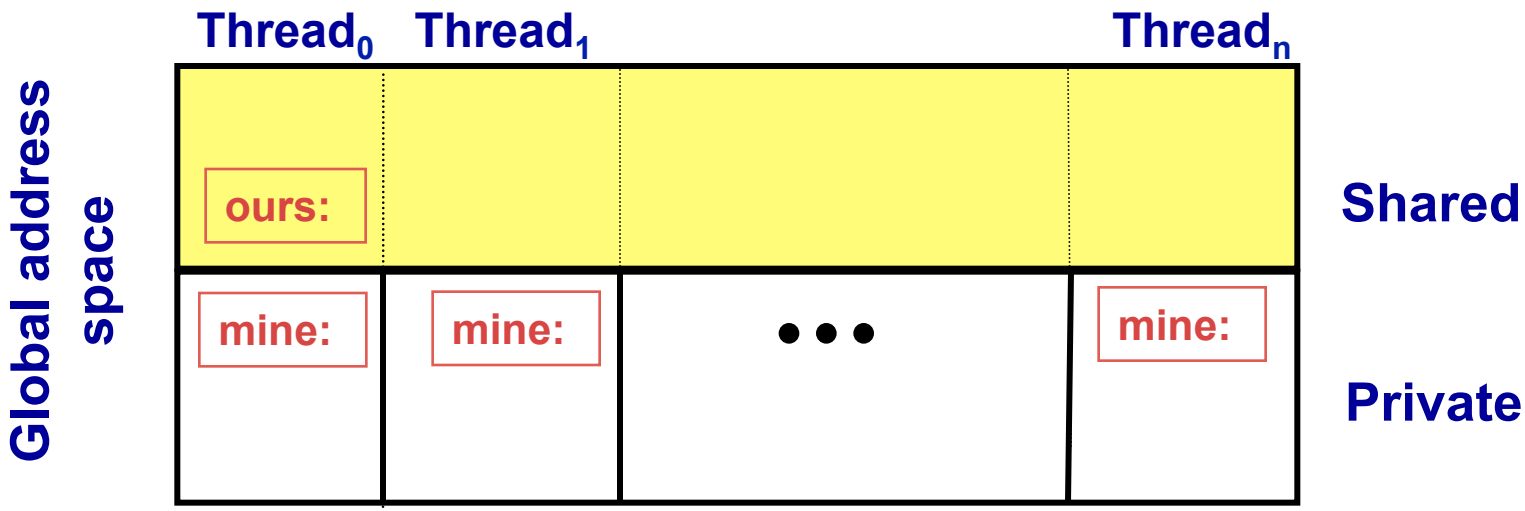
Shared vs. Private Variables

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

```
shared int ours;  // use sparingly: performance
int mine;
```

- Shared variables may not have dynamic lifetime, i.e., may not occur in a function definition, except as static.



Pi in UPC: Shared Memory Style

- Parallel computing of pi, but with a bug

```
shared int hits;
main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
      hits += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
      printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

shared variable to record hits

divide work up evenly

accumulate hits

What is the problem with this program?

UPC Synchronization

- **UPC has two basic forms of barriers:**
  - Barrier: block until all other threads arrive

    ```
    upc_barrier
    ```
  - Split-phase barriers

    ```
    upc_notify;  this thread is ready for barrier
    ```
    do computation unrelated to barrier
    ```
    upc_wait;    wait for others to be ready
    ```
- **UPC also has locks for protecting shared data:**
  - Locks are an opaque type (details hidden):

    ```
    upc_lock_t *upc_global_lock_alloc(void);
    ```
  - Critical region protected by lock/unlock:

    ```
    void upc_lock(upc_lock_t *l)
    void upc_unlock(upc_lock_t *l)
    ```
    use at start and end of critical region

Pi in UPC: Shared Memory Style

- Like pthreads, but use shared accesses judiciously

```
shared int hits;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        my_hits += hit();
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*hits/trials);
}
```

one shared scalar variable

other private variables

create a lock

accumulate hits locally

accumulate across threads

Pi in UPC: Data Parallel Style w/ Collectives

- The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
- Use a reduction for better scalability

```
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ...
    for (i=0; i < my_trials; i++)
       my_hits += hit();
    my_hits =           // type, input, thread, op
       bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*my_hits/trials);
}
```

Berkeley collectives

no shared variables

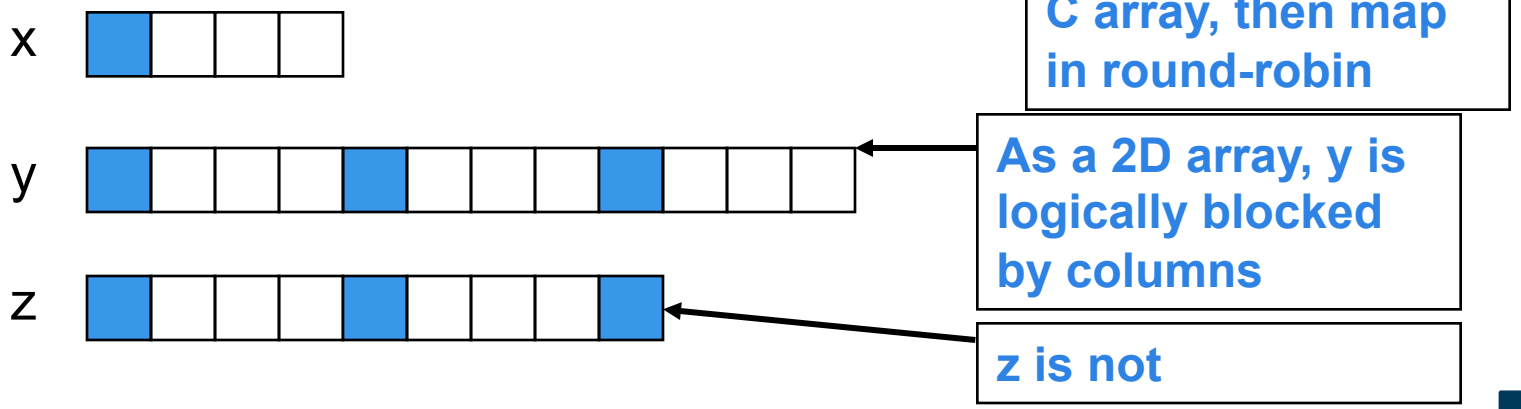
barrier implied by collective

Shared Arrays Are Cyclic By Default

- Shared scalars always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads

```
shared int x[THREADS]      /* 1 element per thread */
shared int y[3][THREADS]   /* 3 elements per thread */
shared int z[3][3]         /* 2 or 3 elements per thread */
```

- In the pictures below, assume THREADS = 4
  - Blue elts have affinity to thread 0



Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
  … declarations an initialization code omitted
  for (i=0; i < my_trials; i++)
    all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++) hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
}
```

all_hits is shared by all processors, just as hits was

update element with local affinity

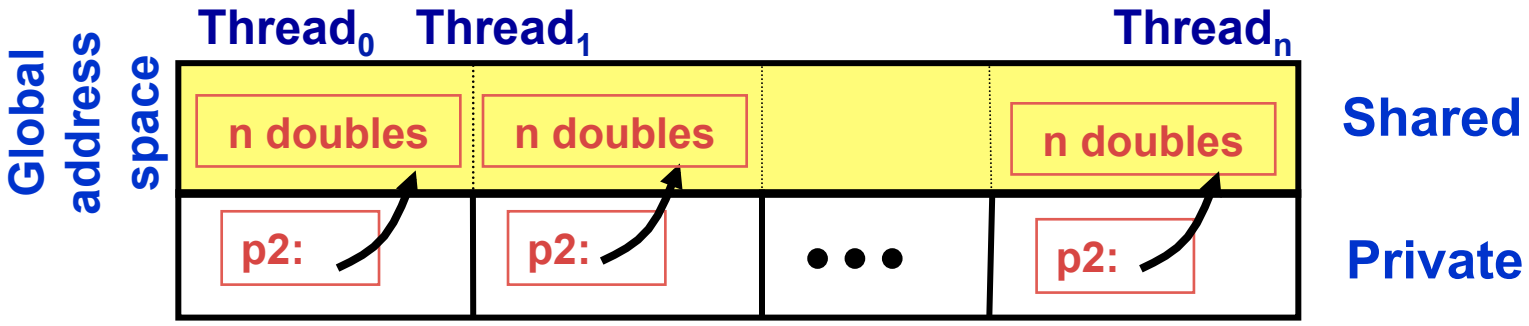
Global Memory Allocation

```
shared void *upc_alloc(size_t nbytes);
```

nbytes : size of memory in bytes

- Non-collective: called by one thread
- The calling thread allocates a contiguous memory space in the shared space with affinity to itself.

```
shared [] double [n] p2 = upc_alloc(n&sizeof(double));
```



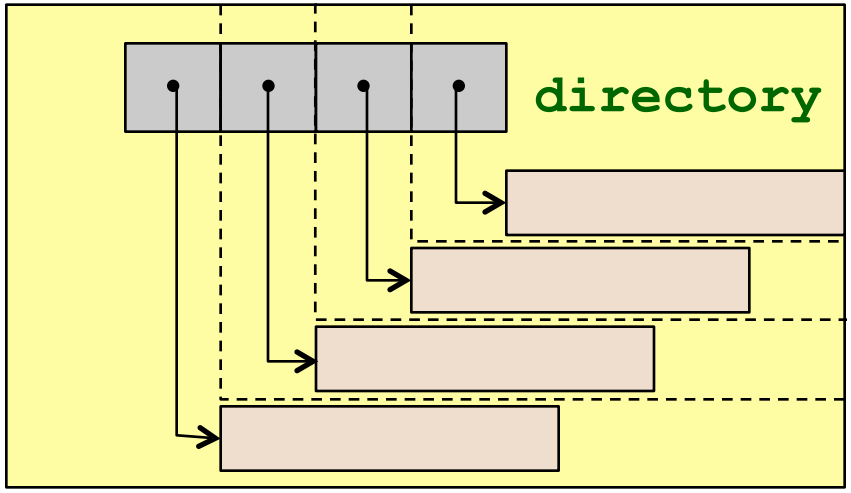

```
void upc_free(shared void *ptr);
```

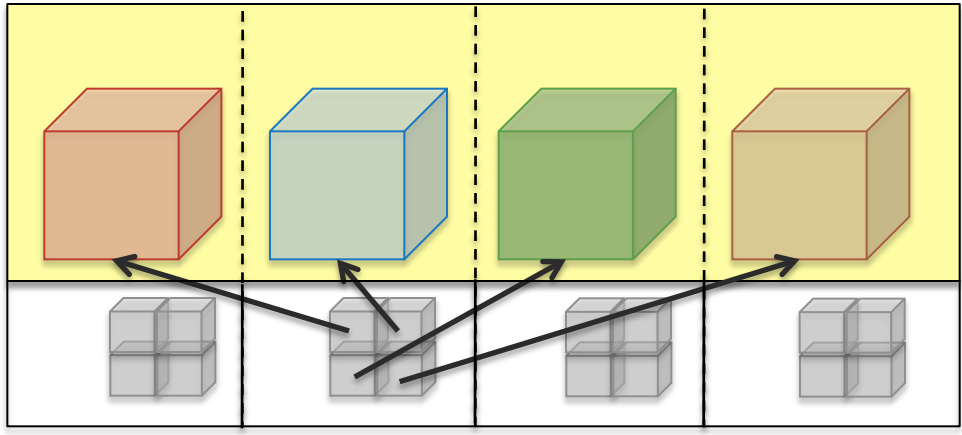
- Non-collective function; frees the dynamically allocated shared memory pointed to by ptr

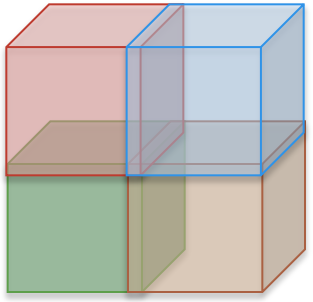
Distributed Arrays Directory Style

- Many UPC programs avoid the UPC style arrays in factor of directories of objects

```
typedef shared [] double *sdblptr;
shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));
```





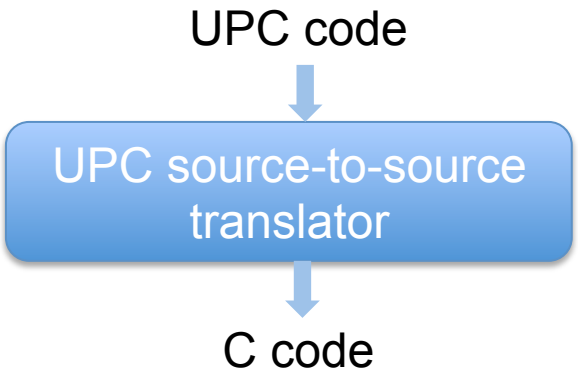


physical and conceptual 3D array layout

- These are also more general:
  - Multidimensional, unevenly distributed
  - Ghost regions around blocks

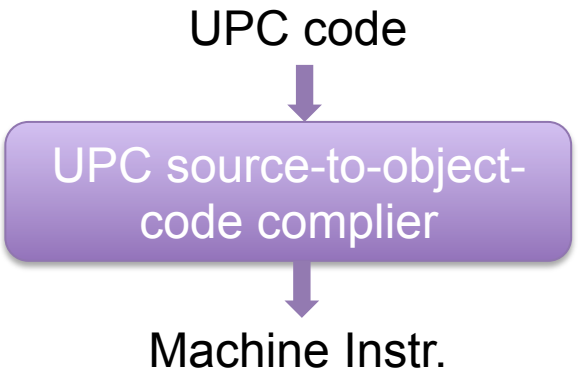
UPC Compiler Implementation

UPC-to-C translator



- Pros: portable, can use any backend C compiler
- Cons: may lose program information between the two compilation phases
- Example: Berkeley UPC

UPC-to-object-code compiler



- Pros: better for implementing UPC specific optimizations
- Cons: less portable
- Example: GCC UPC and most vendor UPC compilers

New in UPC 1.3 Non-blocking Bulk Operations

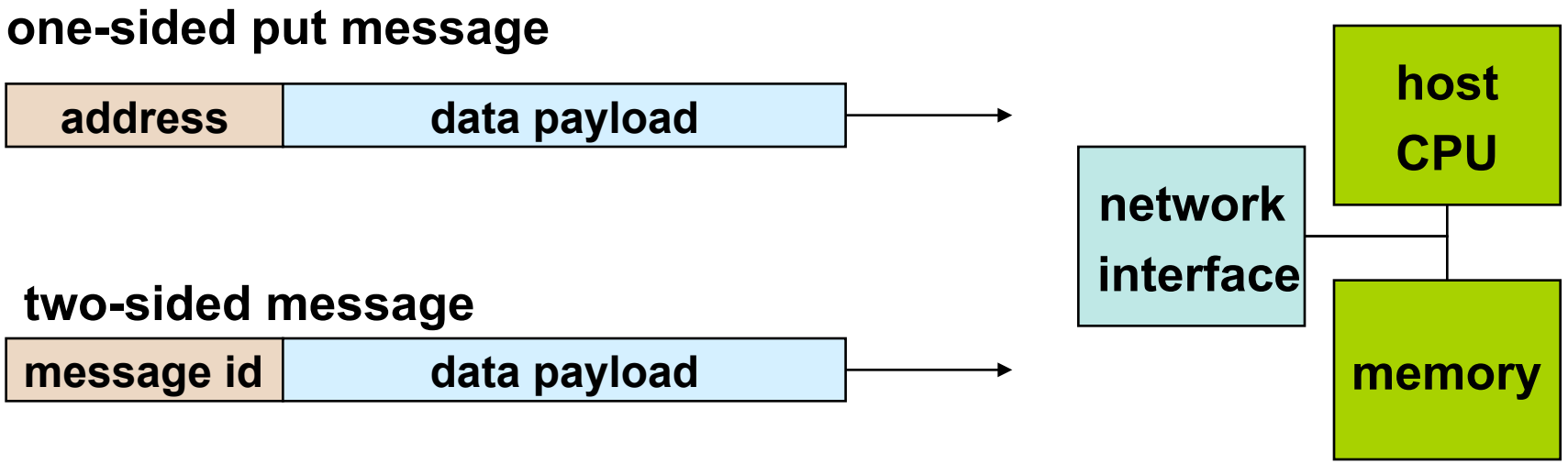
Important for performance:

- **Communication overlap with computation**
- **Communication overlap with communication (pipelining)**
- **Low overhead communication**

```
#include<upc_nb.h>

upc_handle_t h =
upc_memcpy_nb(shared void * restrict dst,
              shared const void * restrict src,
              size_t n);
void upc_sync(upc_handle_t h);          // blocking wait
int upc_sync_attempt(upc_handle_t h); // non-blocking
```

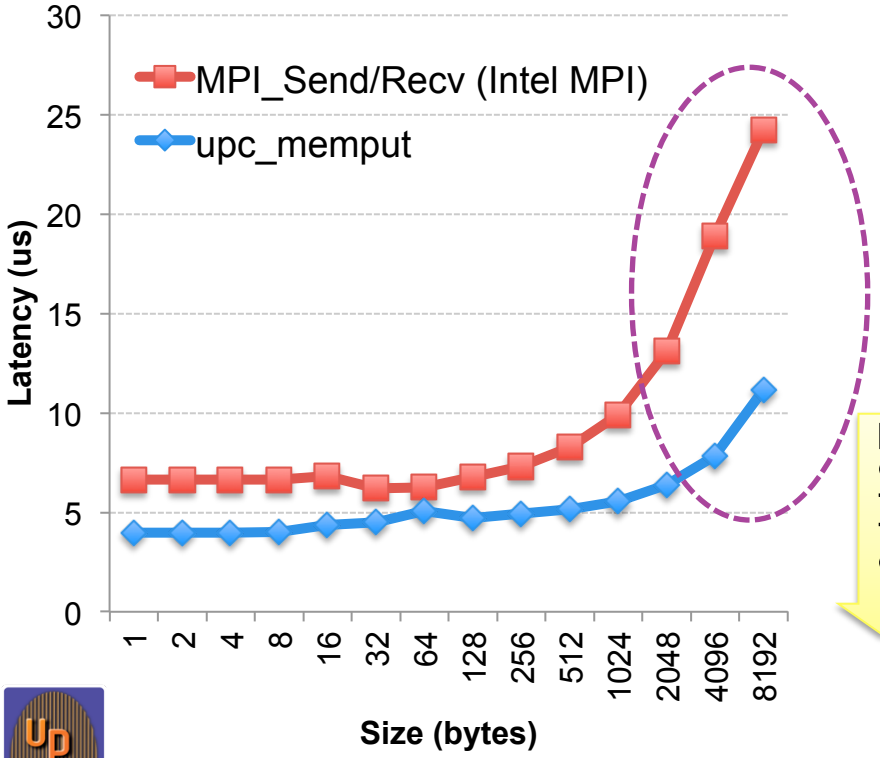
One-Sided in GASNet



- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth

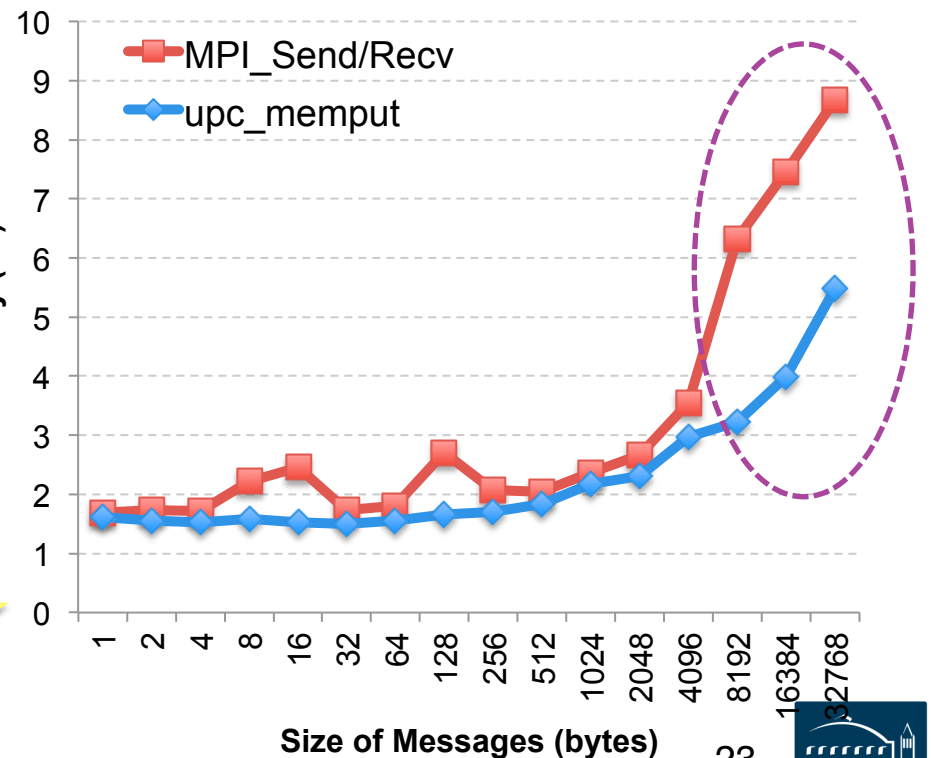
Why Should You Care about PGAS?

Latency between 2 Xeon Phi's via Infiniband

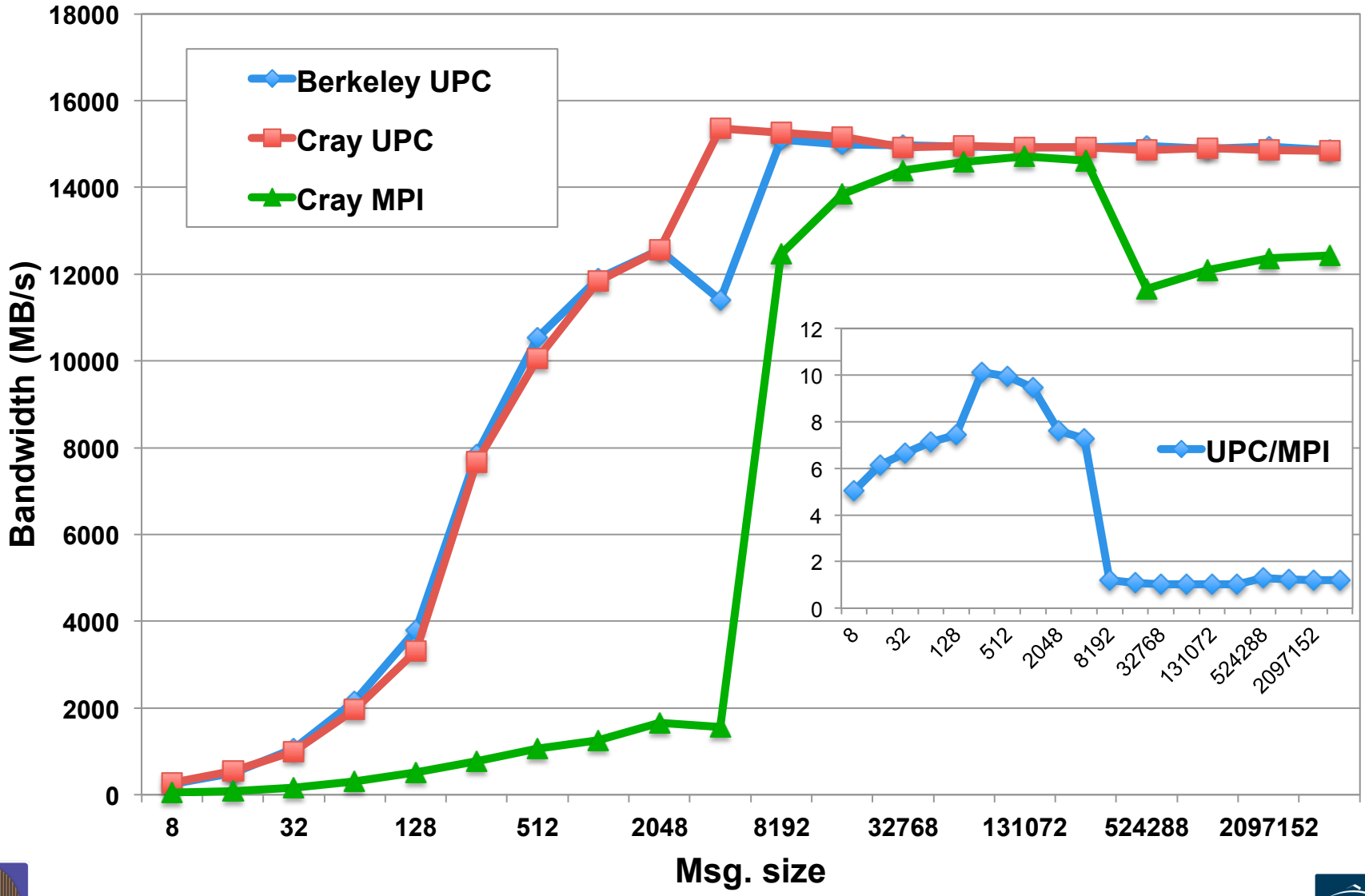


better

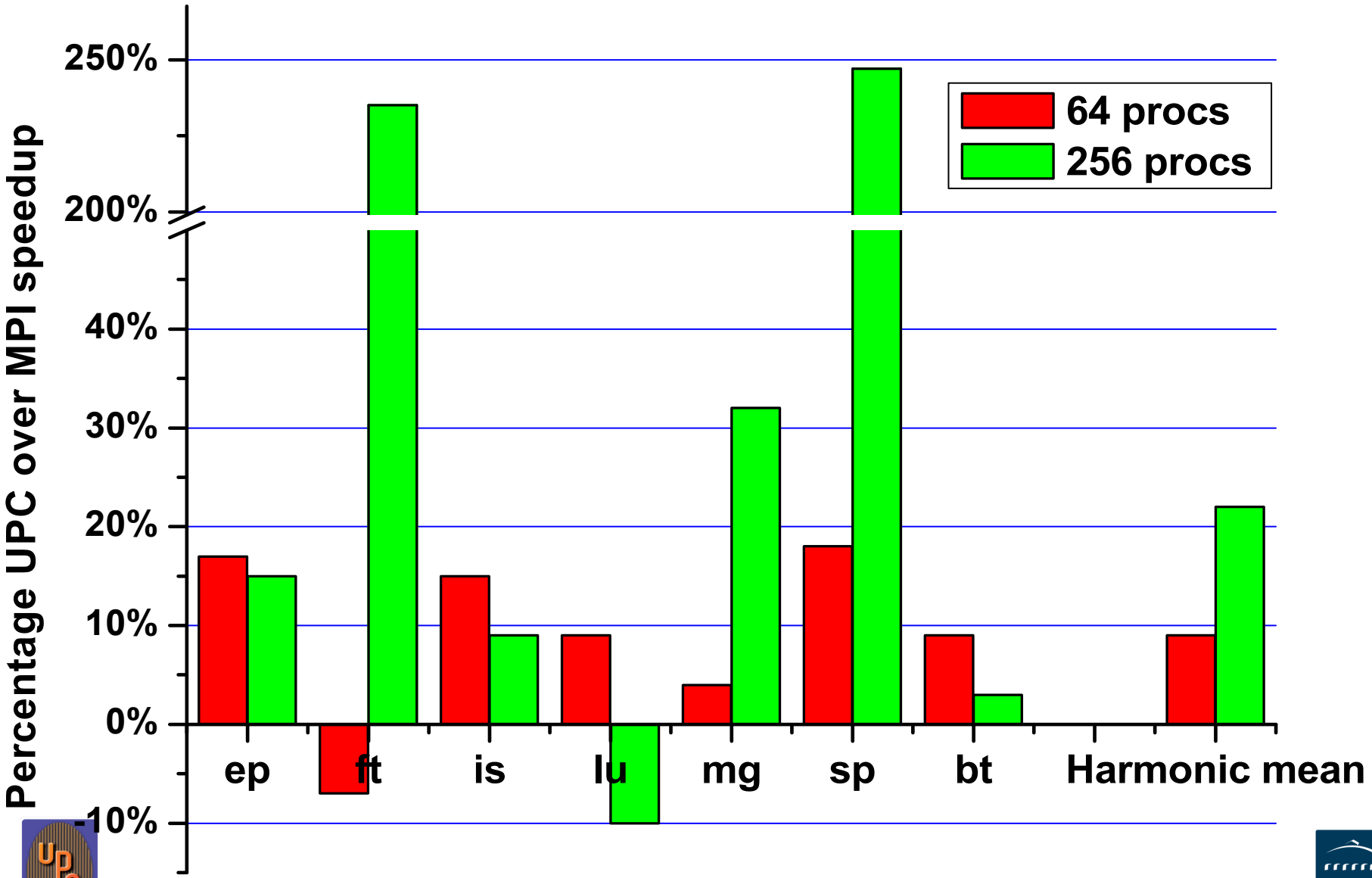
Latency between 2 Intel IvyBridge nodes on NERSC Edison (Cray XC30)



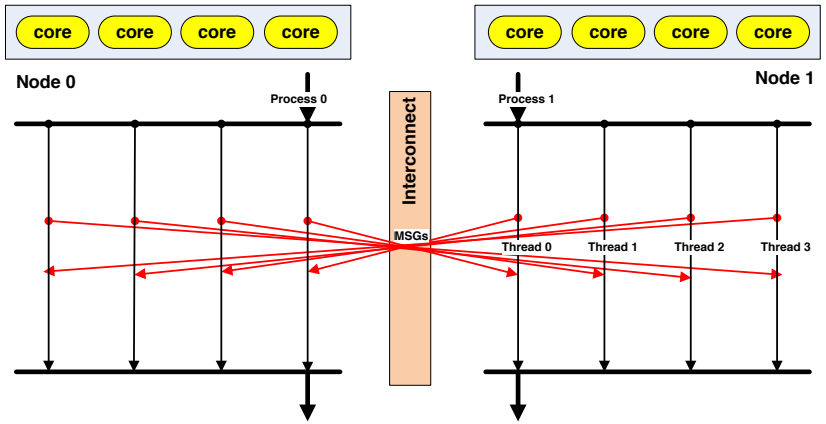
Bandwidths on Cray XE6 (Hopper)

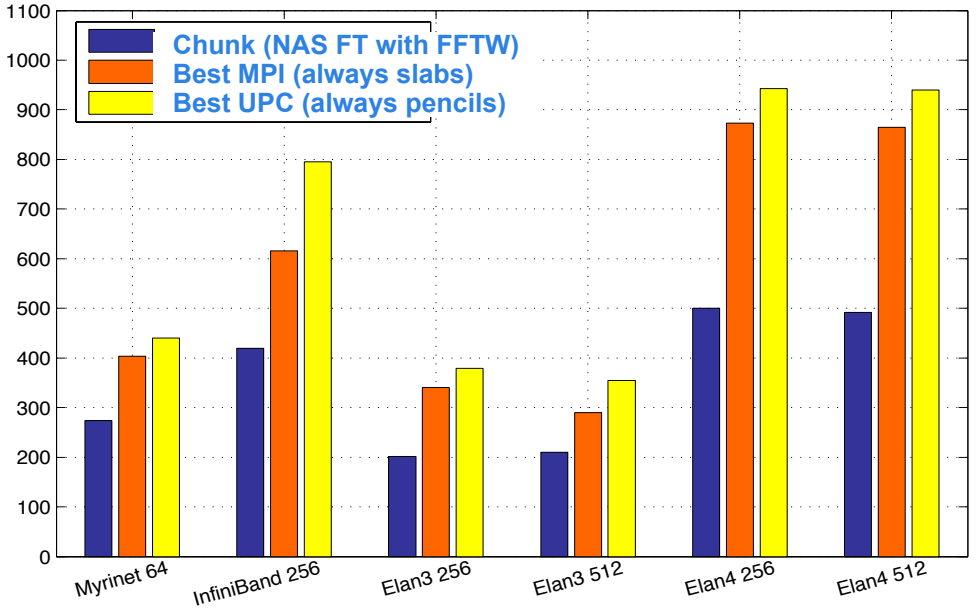


Cray XE6 Application Performance



Machine Challenge #3: Bisection Bandwidth

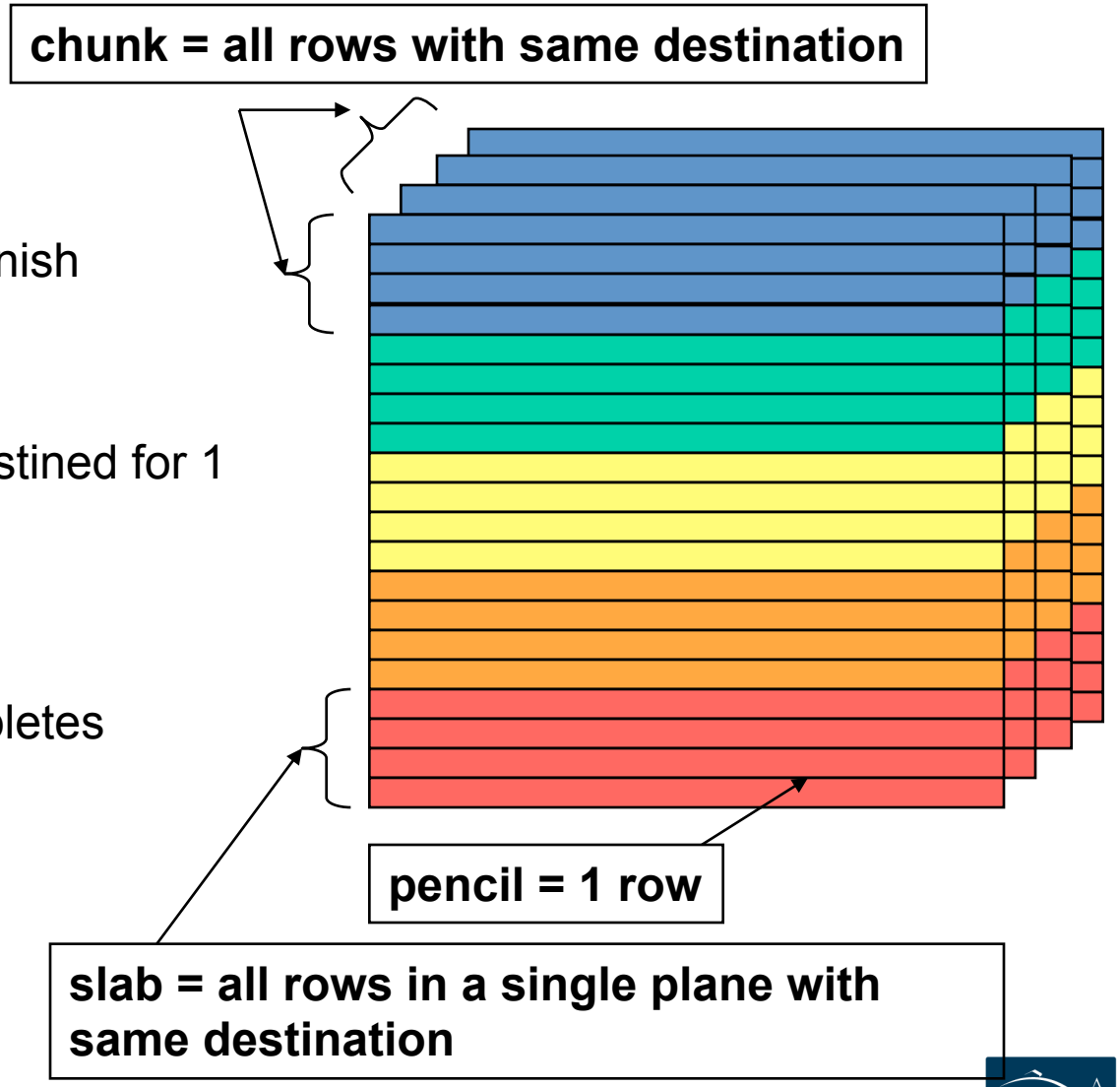




- Avoid congestion at node interface: allow all cores to communicate
- Avoid congestion inside global network: spread communication over longer time period (start early, send often)
- Synchronize only when needed: sometimes fine-grained, sometimes one global barrier (after all incoming counts are reached) is best

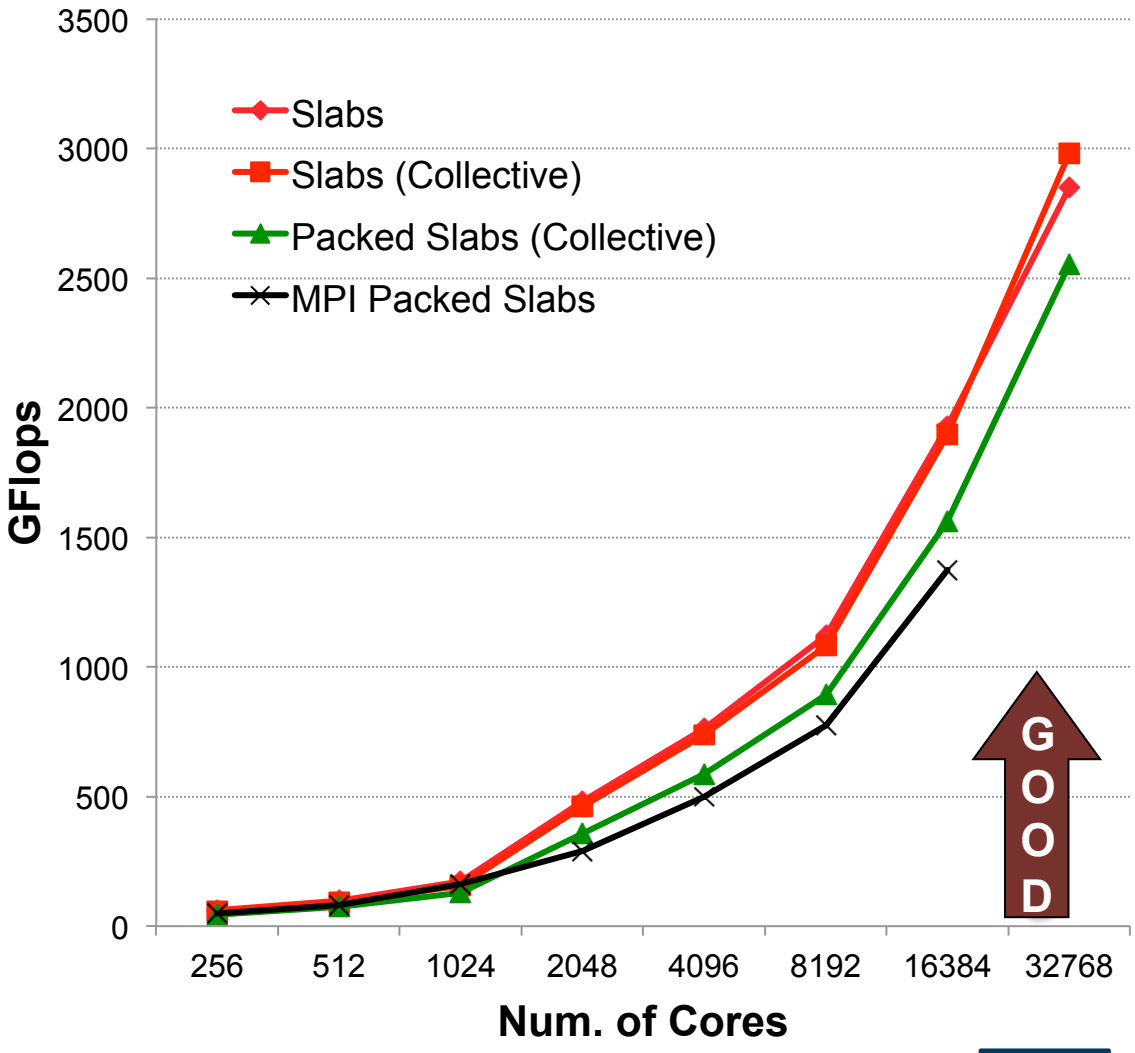
Application Challenge: Fast All-to-All

- Three approaches:
  - **Chunk:**
    - Wait for 2nd dim FFTs to finish
    - Minimize # messages
  - **Slab:**
    - Wait for chunk of rows destined for 1 proc to finish
    - Overlap with computation
  - **Pencil:**
    - Send each row as it completes
    - Maximize overlap and
    - Match natural layout



FFT Performance on BlueGene/P

- **UPC implementation outperforms MPI**
- **Both use highly optimized FFT library on each node**
- **UPC version avoids send/receive synchronization**
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth



BERKELEY LAB
Lawrence Berkeley National Laboratory

UPC 1.3 Atomic Operations

- More efficient than using locks when applicable

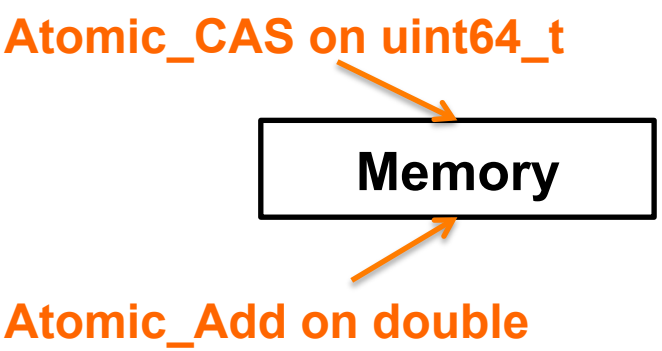
```
upc_lock();
update();
upc_unlock();
```

vs

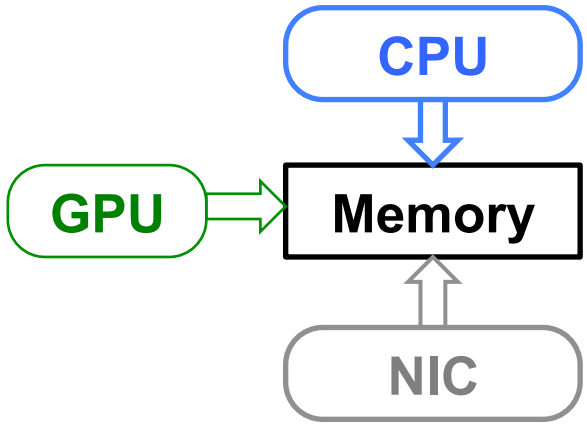
```
atomic_update();
```

- Hardware support for atomic operations are available, *but*

Only support limited operations on a subset of data types. e.g.,



Atomic ops from different processors *may not* be atomic to each other

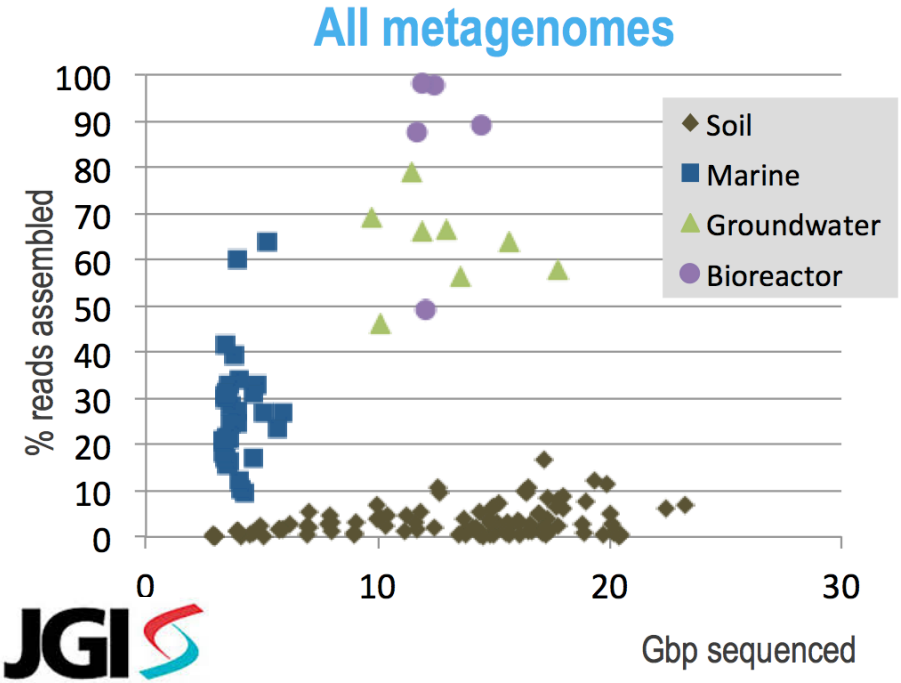


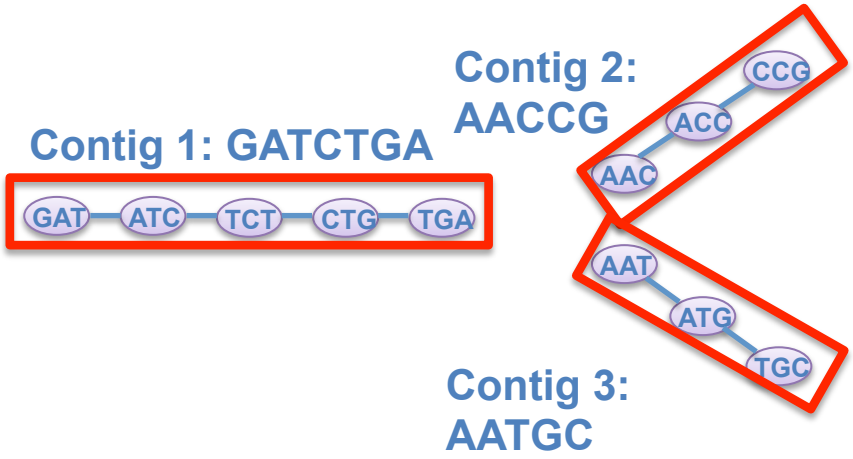
Application Challenge: Random Access to Large Memory

- Expand the class of Exascale applications to those involving random access to large "shared" memory
  - Hash tables
  - Graph algorithms

"Big Data" problems?

- Problems that currently "require" shared memory
- Genome assembly example





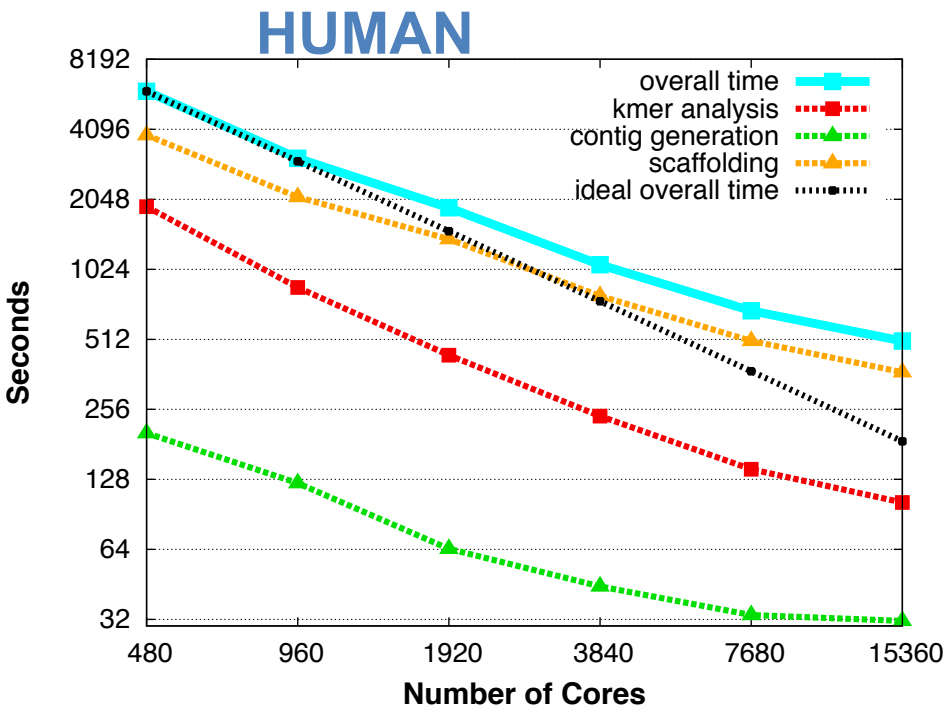
JGI

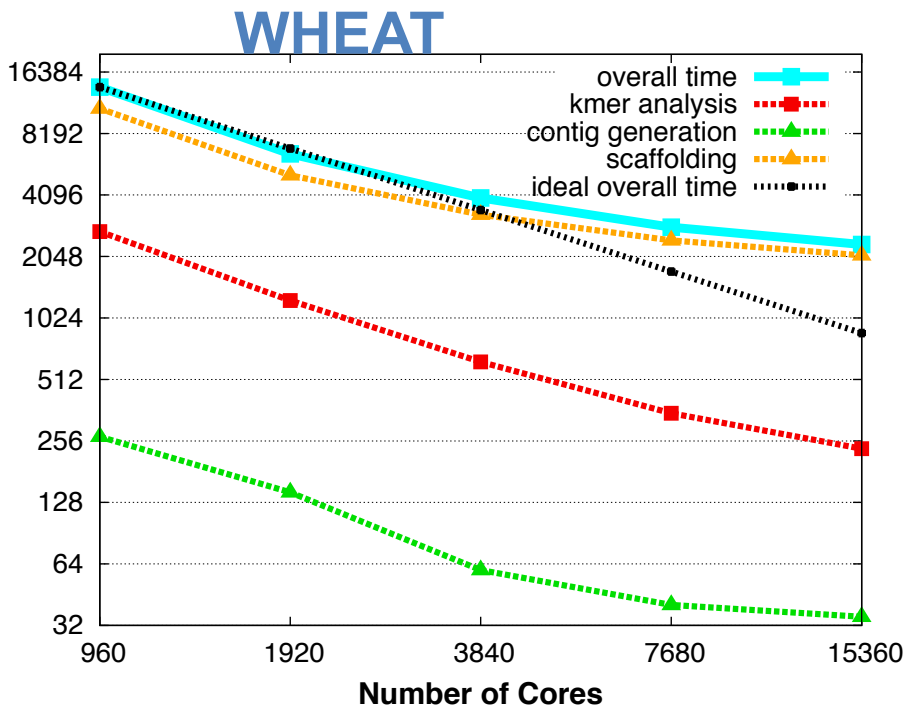
BERKELEY LAB
Lawrence Berkeley National Laboratory

HipMer (High Performance Meraculous) Assembly Pipeline

Distributed Hash Tables in PGAS

- Remote Atomics, Dynamic Aggregation
- Software Caching (sometimes)
- Clever algorithms (bloom filters, locality-aware hashing)





Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Lenny Oliker, Dan Rokhsar, and Kathy Yelick. HipMer: An Extreme-Scale De Novo Genome Assembler, SC'15

BERKELEY LAB
Lawrence Berkeley National Laboratory

UPC++

Led by Yili Zheng (LBNL)
with Amir Kamil (U Mich)

And host of others: Paul Hargrove, Dan Bonachea, John Bachan,

DEGAS is a DOE-funded X-Stack with Lawrence Berkeley National Lab, Rice Univ., UC Berkeley, and UT Austin.

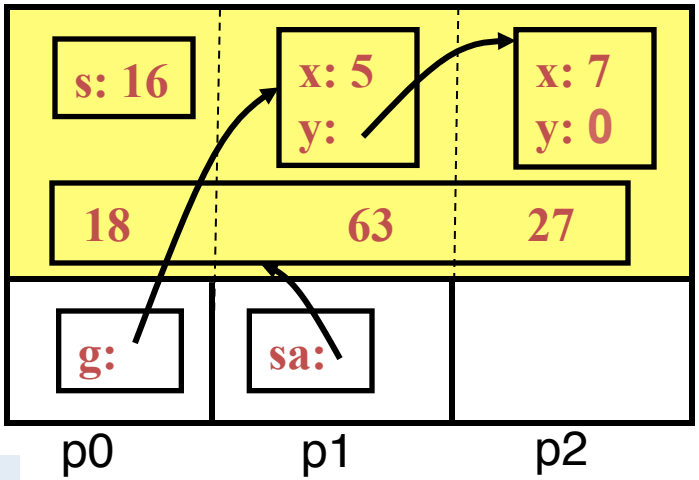
BERKELEY LAB
Lawrence Berkeley National Laboratory

UPC++: PGAS with "Mixins"

- UPC++ uses templates (no compiler needed)

```
shared_var<int> s;
global_ptr<LLNode> g;
shared_array<int> sa(8);
```

- Default execution model is SPMD, but



- Remote methods, async

```
async(place) (Function f, T1 arg1,…);
wait();      // other side does poll();
```

- Research in teams for hierarchical algorithms and machines

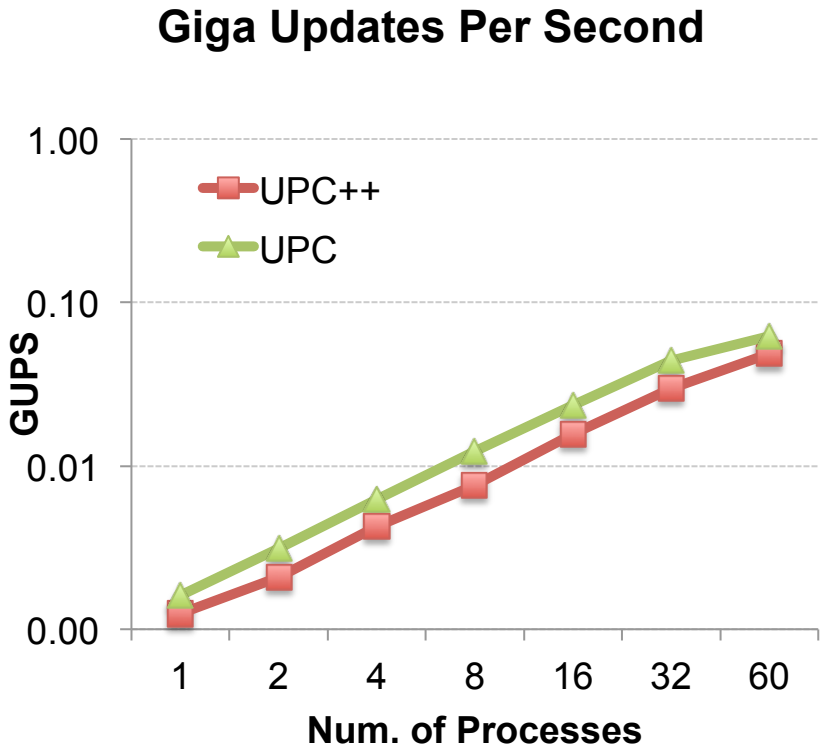
```
teamsplit (team) { ... }
```

- Interoperability is key; UPC++ can be use with OpenMP or MPI

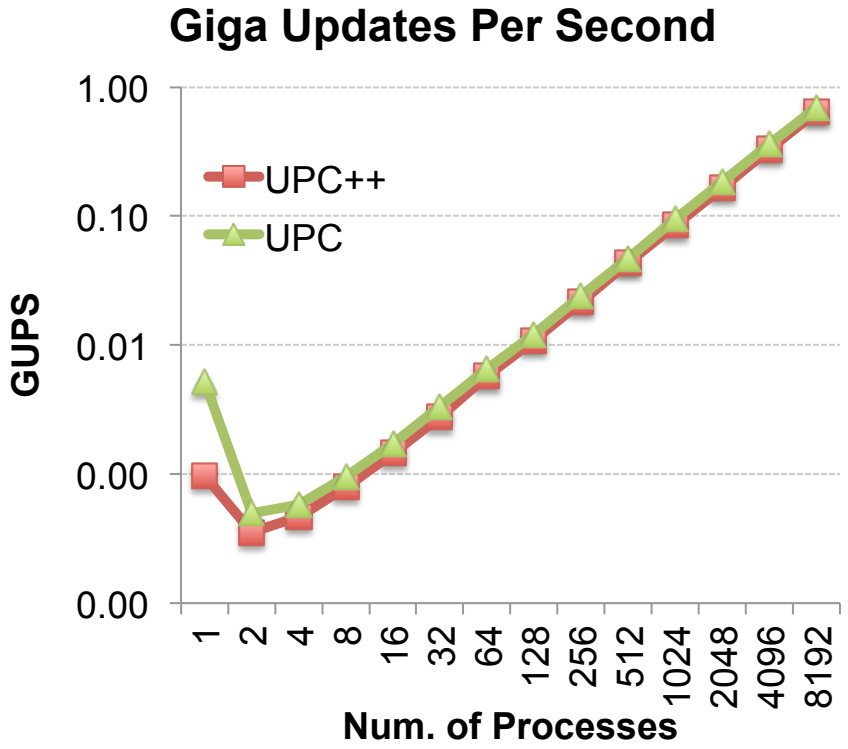
UPC++ Performance Close to UPC

GUPS (fine-grained) Performance on MIC and BlueGene/Q

MIC



BlueGene/Q



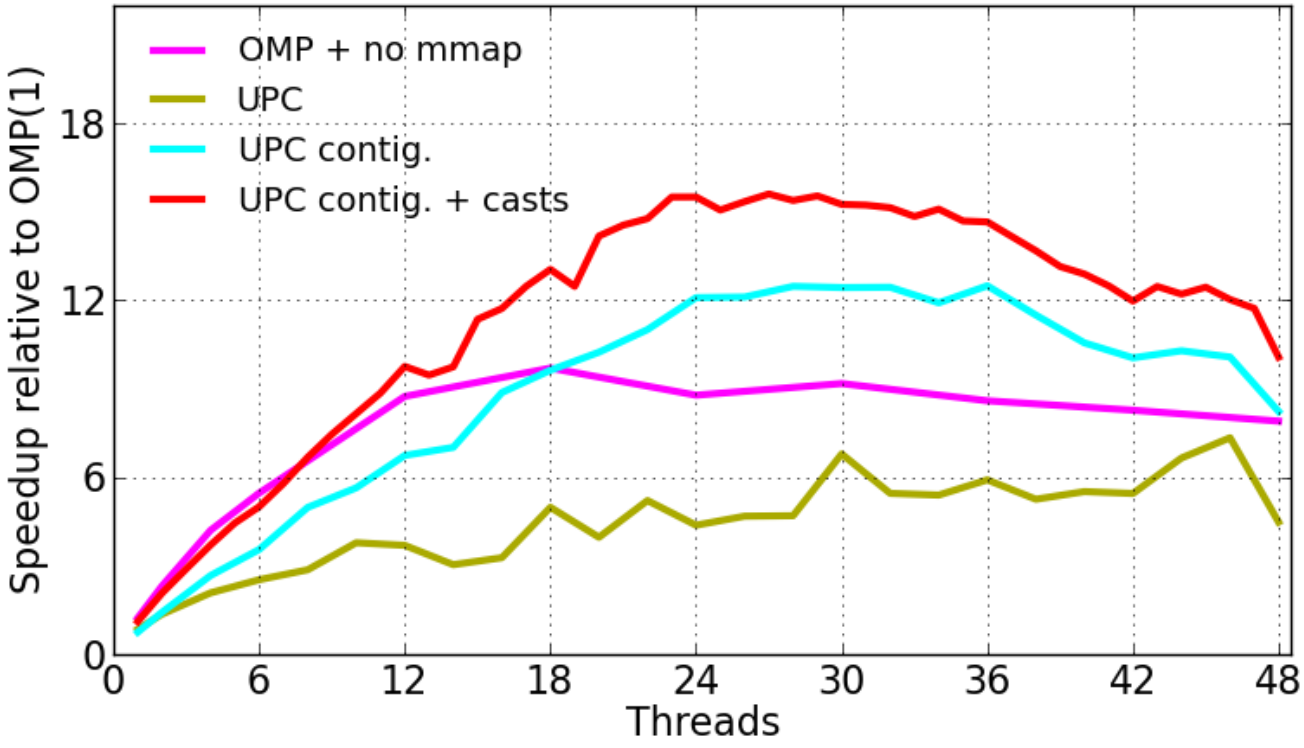
Difference between UPC++ and UPC is about 0.2 μs (~220 cycles)

Locality Control On-Node is Important

Optimizations:

- Blocked vs. cyclic (default) array layout
- Use private pointer to the thread block in shared array

```
double* my_x = (double*)(x + MYTHREAD * BSIZE)
```



Bulk Communication with One-Sided Data Transfers

```
// Copy count elements of T from src to dst
upcxx::copy<T>(global_ptr<T> src,
               global_ptr<T> dst,
               size_t count);

// Non-blocking version of copy
upcxx::async_copy<T>(global_ptr<T> src,
                     global_ptr<T> dst,
                     size_t count);

// Synchronize all previous asyncs
upcxx::async_wait();
```

Similar to *upc_memcpy_nb* extension in UPC 1.3

Dynamic Global Memory Management

- Global address space pointers (pointer-to-shared)

```
global_ptr<data_type> ptr;
```

- Dynamic shared memory allocation

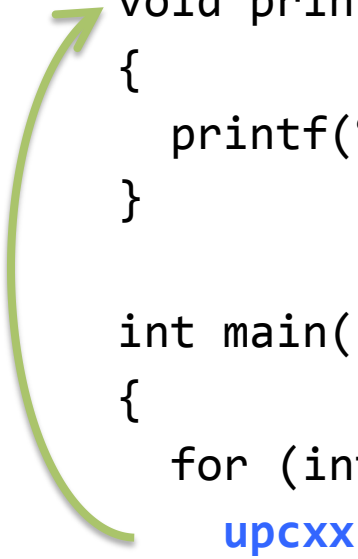
```
global_ptr<T> allocate<T>(uint32_t where,
                          size_t count);
void deallocate(global_ptr<T> ptr);
```

Example: allocate space for 512 integers on rank 2

```
global_ptr<int> p = allocate<int>(2, 512);
```

Remote memory allocation is not available in MPI-3, UPC or SHMEM.

Async Task Example



```
#include <upcxx.h>

void print_num(int num)
{
  printf("myid %u, arg: %d\n", MYTHREAD, num);
}

int main(int argc, char **argv)
{
  for (int i = 0; i < upcxx::ranks(); i++) {
    upcxx::async(i)(print_num, 123);
  }
  upcxx::async_wait(); // wait for all remote tasks to complete
  return 0;
}
```

Async with C++11 Lambda Function

```
for (int i = 0; i < upcxx::ranks(); i++) {
  // spawn a task expressed by a lambda function
  upcxx::async(i)([] (int num)
                  { printf("num: %d\n”, num); },
                  1000+i); // argument to the λ function
}
upcxx::async_wait(); // wait for all tasks to finish
```

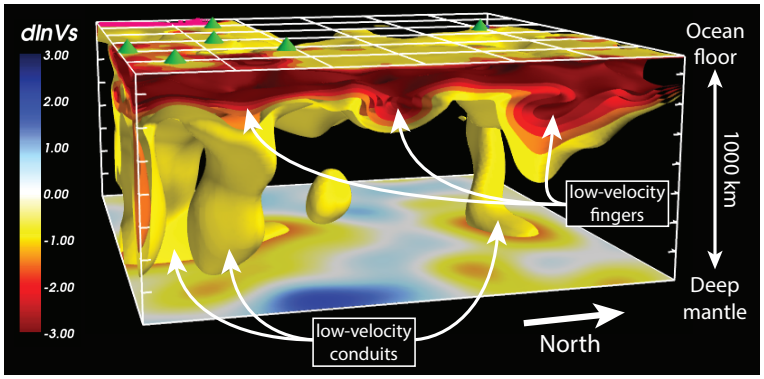
mpirun –n 4 ./test_async

Output:

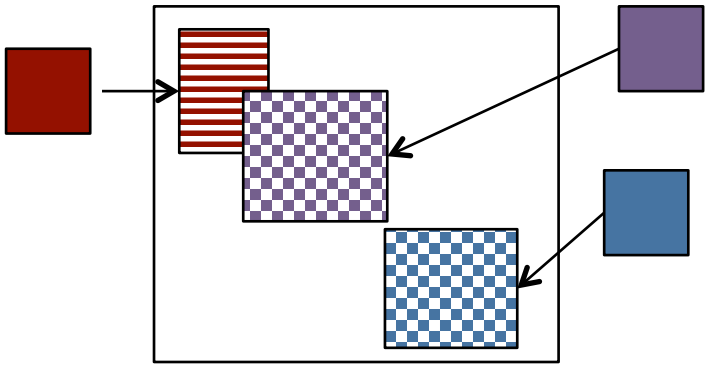
```
num:  1000
num:  1001
num:  1002
num:  1003
```

Function arguments and lambda-captured values must be std::is_trivially_copyable.

Application Challenge: Data Fusion in UPC++



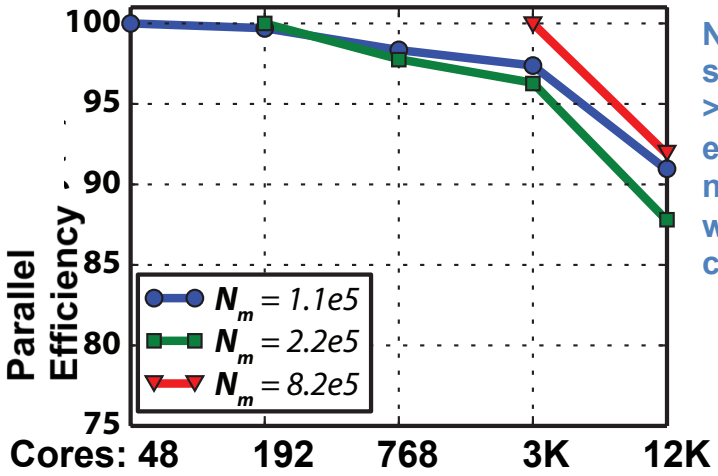
- Seismic modeling for energy applications "fuses" observational data into simulation
- With UPC++, can solve larger problems



Distributed Matrix Assembly

- Remote asyncs with user-controlled resource management
- Remote memory allocation
- Team idea to divide threads into injectors / updaters
- 6x faster than MPI 3.0 on 1K nodes

→ Improving UPC++ team support

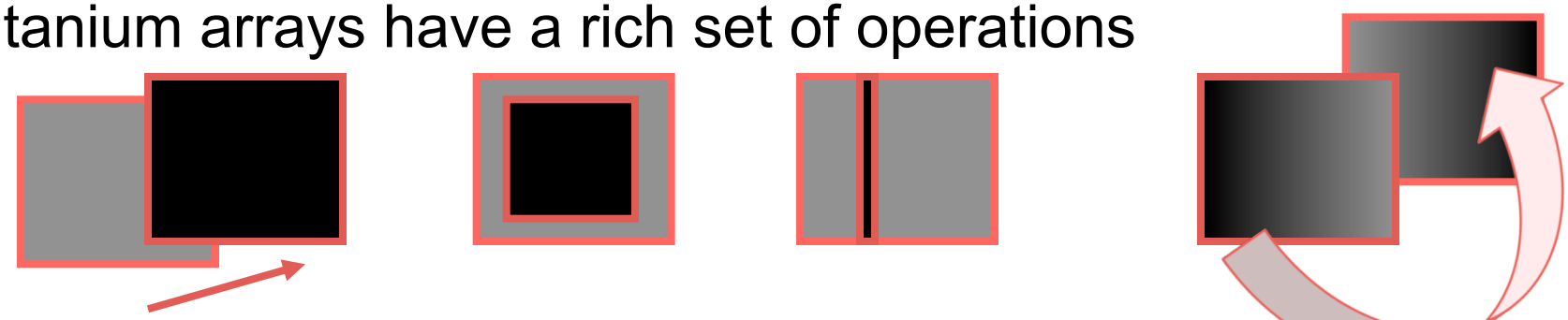


Note scale: >85% efficient in worst case

French and Romanowicz use code with UPC++ phase to compute *first ever* whole-mantle global tomographic model using numerical seismic wavefield computations (F & R, 2014, GJI, extending F et al., 2013, Science). See F et al, IPDPS 2015 for parallelization overview.

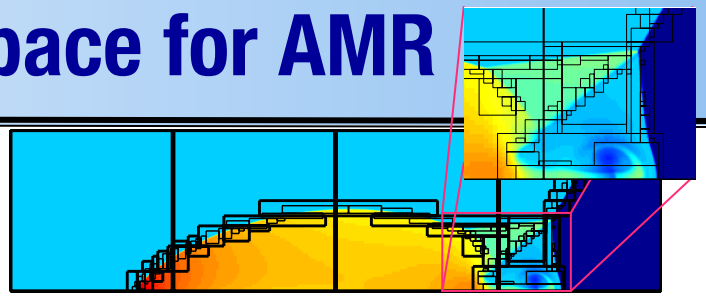
Multidimensional Arrays in UPC++ (and Titanium)

- Titanium arrays have a rich set of operations



- None of these modify the original array, they just create another view of the data in that array
- You create arrays with a RectDomain and get it back later using A.domain() for array A
  - A Domain is a set of points in space
  - A RectDomain is a rectangular one
- Operations on Domains include +, -, * (union, different intersection)

Arrays in a Global Address Space for AMR

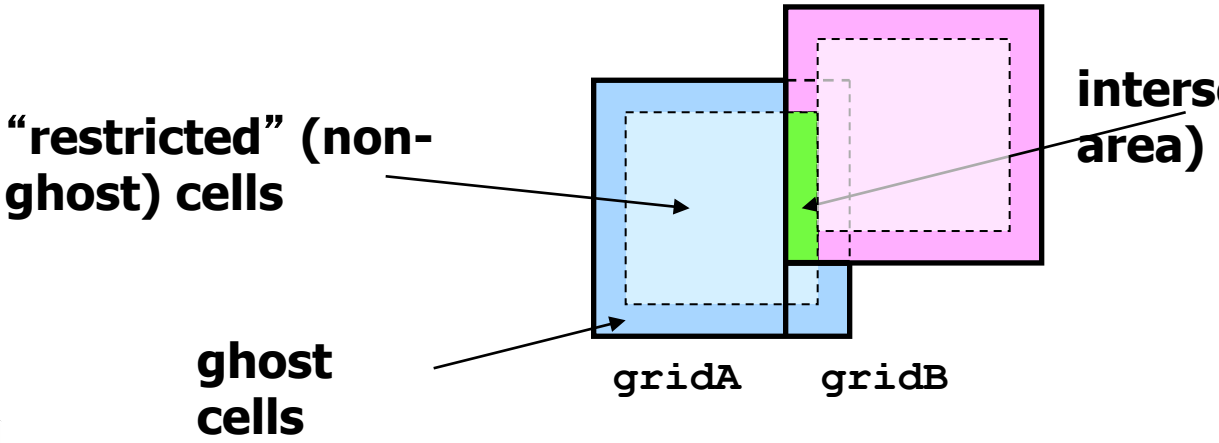


- Key features of UPC++ arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to iterate over interior:

```
foreach (idx, gridB.shrink(1).domain())
```

- Array copies automatically work on intersection

```
gridB.copy(gridA.shrink(1));
```

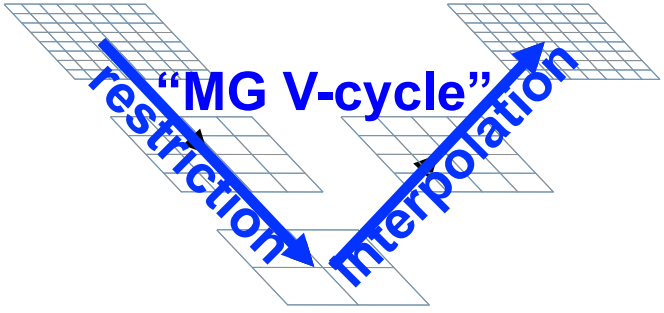


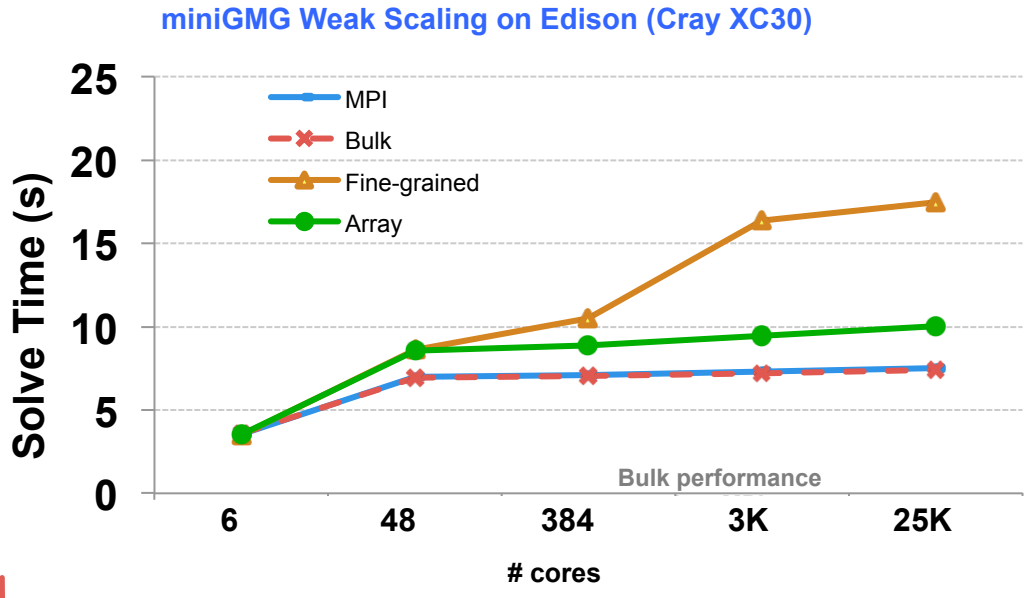
Useful in grid computations including AMR

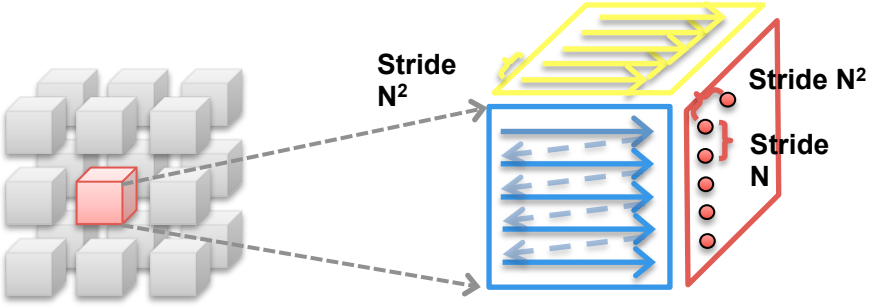
UPC++ arrays based on Titanium Arrays

Mini-GMG in UPC++ uses high level array library for Productivity and Performance

miniGMG proxy for Multigrid solver in combustion, etc.







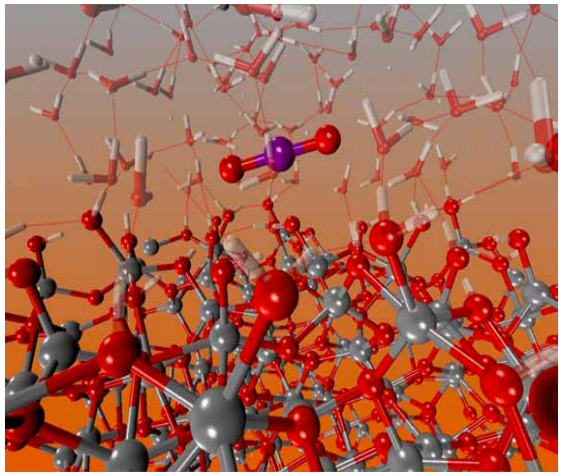
UPC++ arrays are convenient and optimize strided data accesses automatically

- "Fine-grained" like OpenMP
- "Bulk" like MPI with 1-sided communication;
- "Array" version uses multi-dimensional array constructs for productivity and ~MPI performance
- Future runtime optimizations should close Array/Bulk gap

DEGAS

BERKELEY LAB

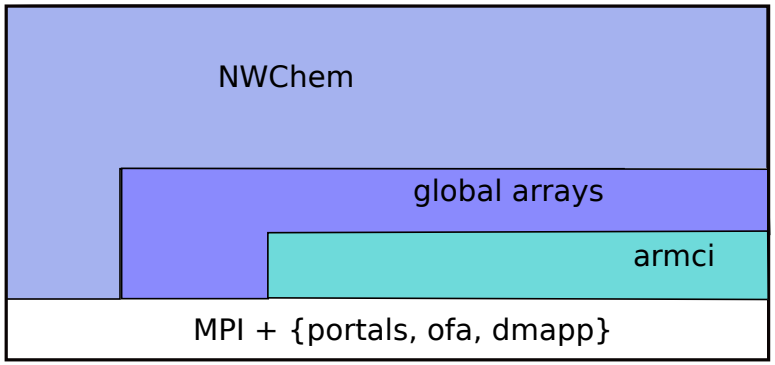
NWChem on GASNet

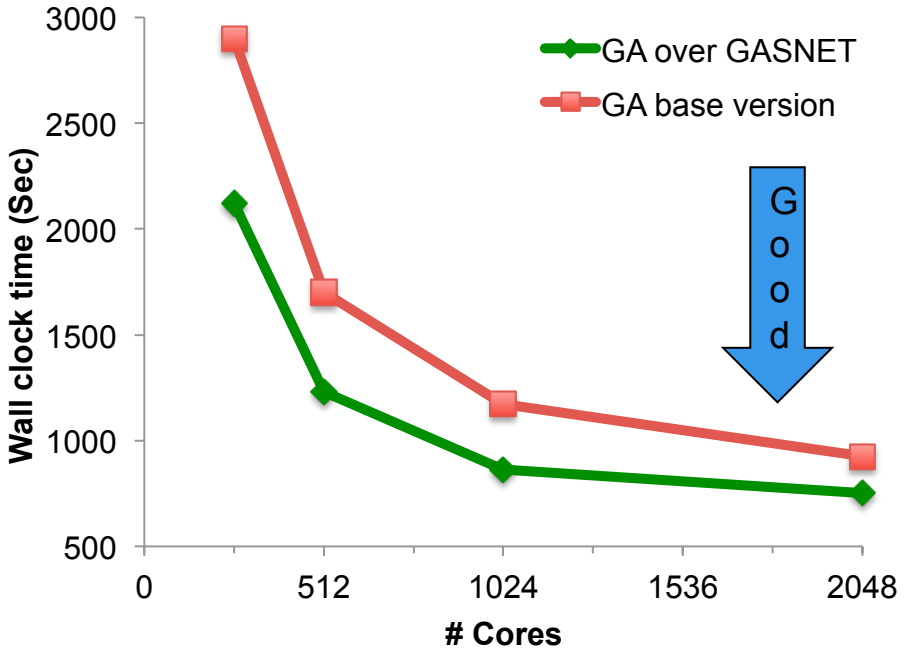


credit:nwchem-sw.org

- **Production chemistry code**
  - 60K downloads world wide
  - 200-250 scientific application publications per year
  - Over 6M LoC, 25K files

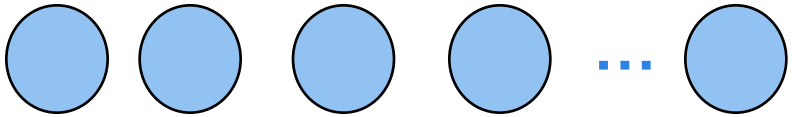
- **New version on GASNet for**
  - Improved performance
  - Portability with other PGAS





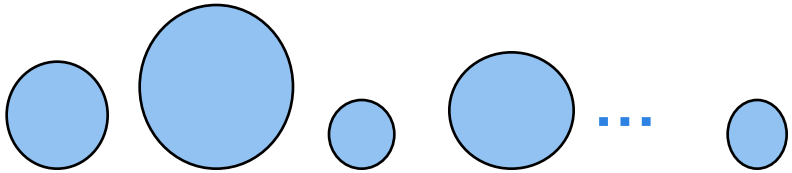
Application Challenge: Dynamic Load Balancing

- **Static:** Equal size tasks



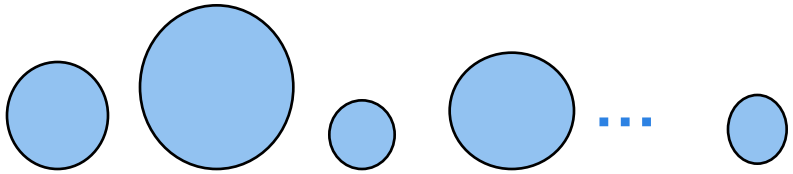
Regular meshes, dense matrices, direct n-body

- **Semi-Static:** Tasks have different but estimable times



Adaptive and unstructured meshes, sparse matrices, tree-based n-body, particle-mesh methods

- **Dynamic:** Times are not known until mid-execution



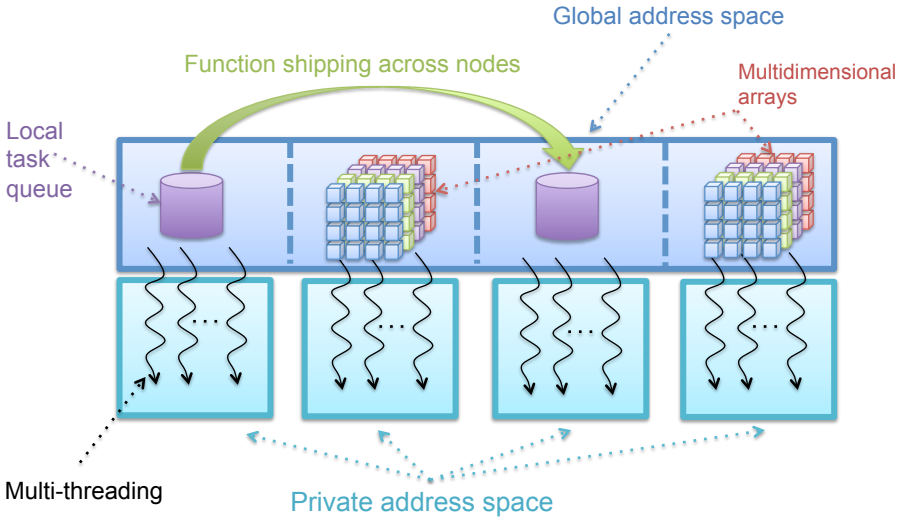
Search (UTS), irregular boundaries, subgrid physics, unpredictable machines

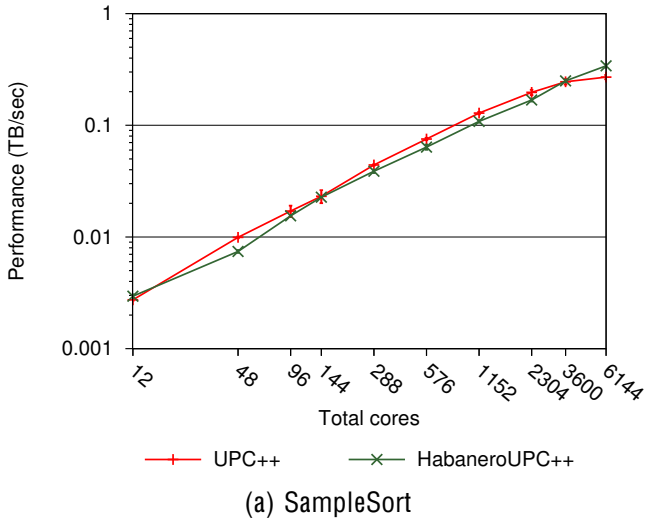
Dynamic (on-the-fly) useful when:

Load imbalance penalty > communication to balance

Load balancing can't solve lack of parallelism

Application Challenge: Dynamic Load Balance

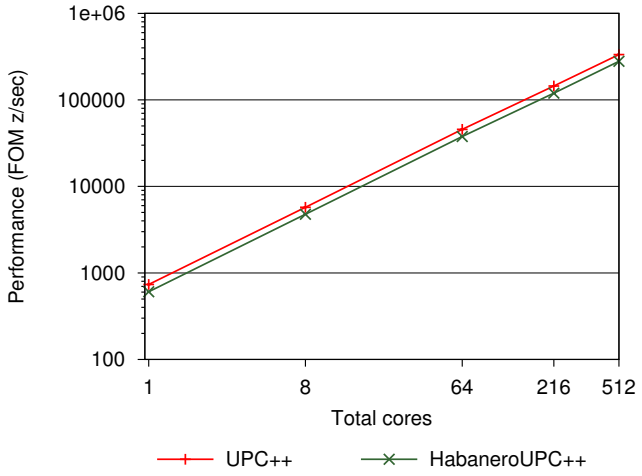




(a) SampleSort

- Dynamic tasking option in UPC++
  - Demonstrated with library version of Habanero
  - Combines with remote async

→ Dynamic load balancing library for domain-specific runtime in UPC++

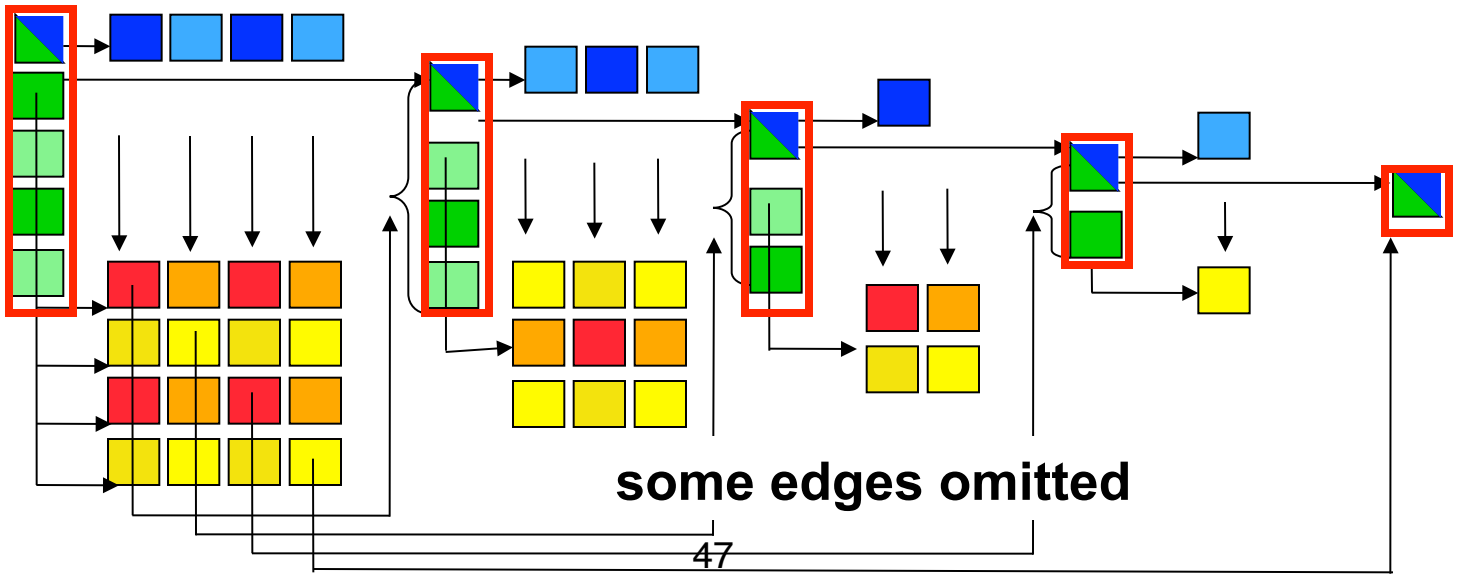


(b) LULESH

Beyond Put/Get: Event-Driven Execution

- DAG Scheduling in a distributed (partitioned) memory context
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - "memory constrained" lookahead

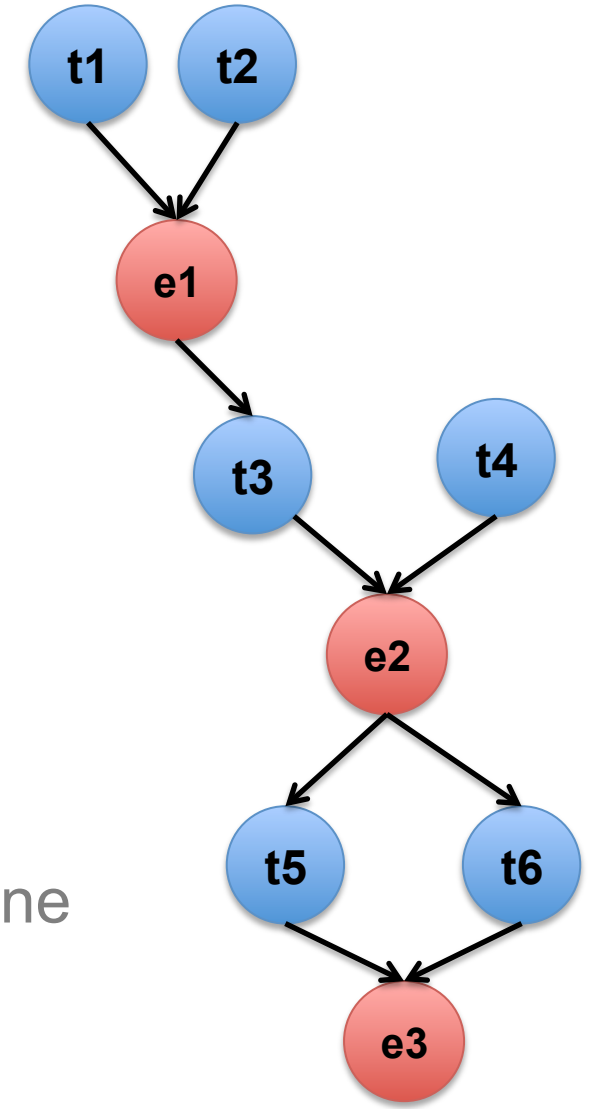
Uses a Berkeley extension to UPC to remotely synchronize



Example: Building A Task Graph

```
using namespace upcxx;
event e1, e2, e3;

async(P1, &e1)(task1);
async(P2, &e1)(task2);
async_after(P3, &e1, &e2)(task3);
async(P4, &e2)(task4);
async_after(P5, &e2, &e3)(task5);
async_after(P6, &e2, &e3)(task6);
async_wait(); // all tasks will be done
```

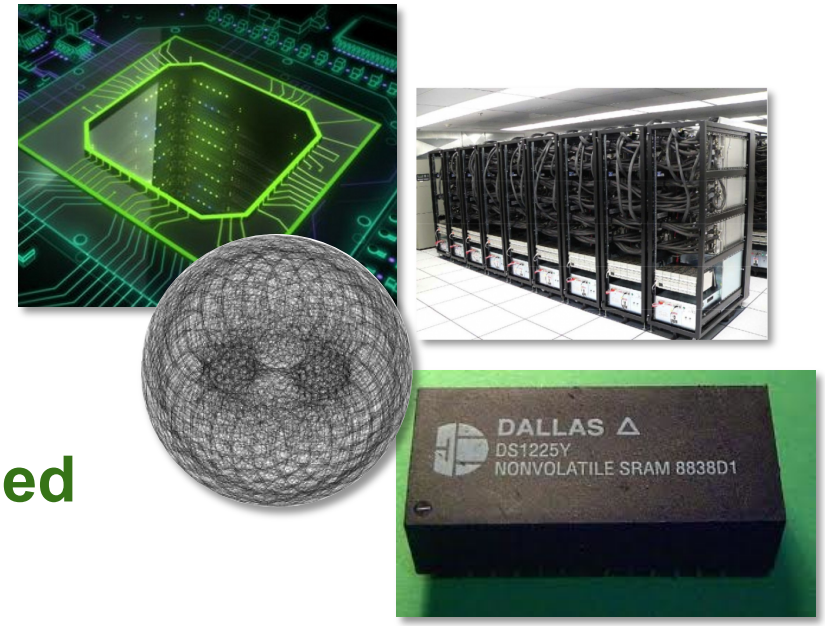


One-sided communication works everywhere

PGAS programming model

```
*p1 = *p2 + 1;
A[i] = B[i];

upc_memput(A,B,64);
```



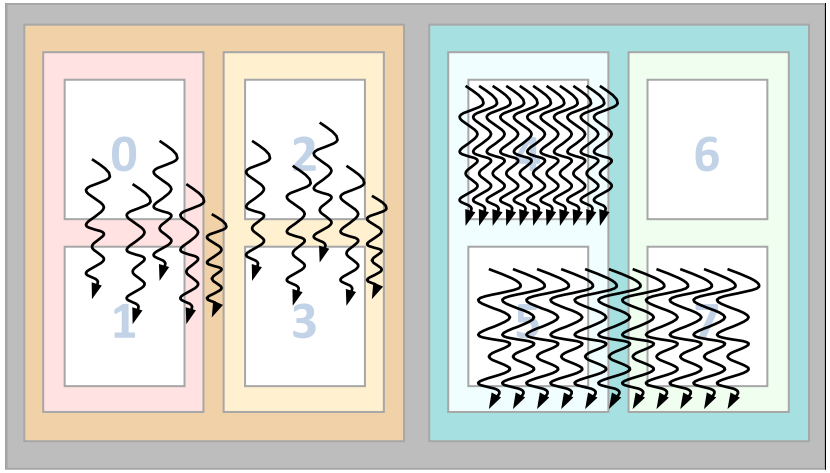
It is implemented using one-sided communication: put/get

Support for one-sided communication (DMA) appears in:

- Fast one-sided network communication (RDMA, Remote DMA)
- Move data to/from accelerators
- Move data to/from I/O system (Flash, disks,..)
- Movement of data in/out of local-store (scratchpad) memory

BERKELEY LAB
Lawrence Berkeley National Laboratory

Hierarchical machines and Applications



- **Hierarchical memory model may be necessary (what to expose vs hide)**
- **Two approaches to supporting the hierarchical control**

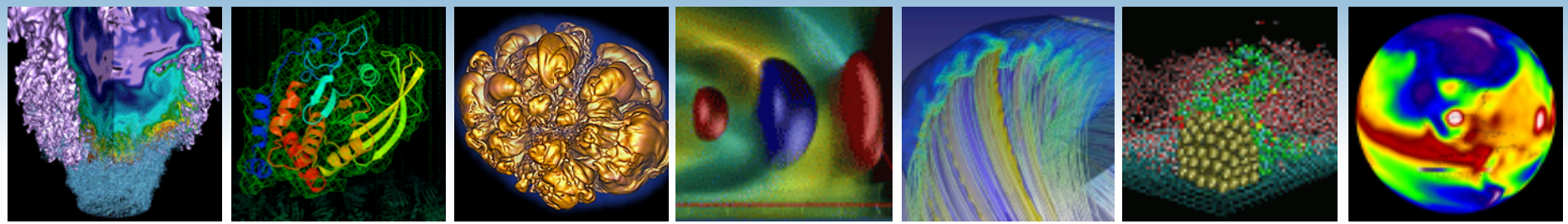
- **Option 1: Dynamic parallelism creation**
  - Recursively divide until… you run out of work (or hardware)
  - Runtime needs to match parallelism to hardware hierarchy
- **Option 2: Hierarchical SPMD with "Mix-ins" (e.g., UPC++)**
  - Hardware threads can be grouped into units hierarchically
  - Add dynamic parallelism with voluntary tasking on a group
  - Add data parallelism with collectives on a group

Summary

- UPC is a mature language with multiple implementations
  - Cray compiler
  - gcc version of UPC: http://www.gccupc.org/
  - Berkeley compiler: http://upc.lbl.gov
- Language specification and other documents

  https://code.google.com/p/upc-specification

  https://upc-lang.org
- UPC++
  - Newer “language” under development
  - Adds dynamic parallelism on top of SPMD default
  - Powerful Multi-D arrays
  - Hierarchical parallelism mapped to machine



LBNL / UCB Collaborators

- Yili Zheng
- Amir Kamil*
- Paul Hargrove
- Eric Roman
- Dan Bonachea*
- Khaled Ibrahim
- Costin Iancu
- Michael Driscoll
- Evangelos Georganas
- Penporn Koanantakool
- Steven Hofmeyr*
- Leonid Oliker
- John Shalf

** Former LBNL/UCB*

- Erich Strohmaier
- Samuel Williams
- Cy Chan
- Didem Unat*
- James Demmel
- Scott French
- Edgar Solomonik*
- Eric Hoffman*
- Wibe de Jong

External collaborators (& their teams!)

- Vivek Sarkar, Rice
- John Mellor-Crummey, Rice
- Mattan Erez, UT Austin

Thanks!